Chapitre 5 : approfondissements sur les listes

A la fin du chapitre précédent, on a découvert un comportement surprenant des listes : si `L` est une liste et qu'on définit `M=L` toute modification de `L` modifiera aussi `M`. Nous avons expliqué ce comportement à partir du fait que le contenu de la case mémoire vers laquelle pointe le nom `L` est une liste d'adresses et pas une liste de valeurs.

1 Comment fabriquer une copie plus autonome d'une liste

1.1 La première méthode : la copie par extraction

On a vu (chap. 2, extraction, slicing) que si `L` est une liste comme celle de notre exemple, la commande `L[0:2]` va fabriquer une nouvelle liste contenant `L[0] et L[1]`.

Ce qui est intéressant avec cette commande, c'est que la nouvelle liste ainsi créée a un identifiant mémoire différent de celui de `L` donc si on extrait *toutes* les valeur de `L`, on a une copie du *contenu* de `L` qui est cette fois à un endroit différent dans la mémoire.

```
L=[13,4,5,6]
M=L[0:4]
print(id(L))
print(id(M))
```

Du coup cette fois, `M` apparaît bien comme autonome de `L` puisque

```
L[0]=45
print(L)
print(M)
```

Pratique : pour extraire *tout* le contenu d'une liste `L`, commande `L[:]`

1.2 Pourquoi PYTHON appelle-t-il le type de copie précédente une *shallow copy* ?

Shallow copy : copie peu profonde.

La raison est qu'on peut faire des *listes de listes* ! Et dans ce cas... il faut faire un dessin encore plus compliqué de ce qu'il y a en mémoire si :

```
L=[1,2,3]
Grosse_bete=[L,"marcel"]
```

Que se passe-t-il alors si je fais une copie de `Grosse_bete` par la méthode d'extraction ?

```
GB2=Grosse_bete[:]
```

Bien sûr `GB2` a un `id` différent de `Grosse_bete` *mais il contient la même liste d'adresses mémoires.*

Faire un schéma !

Si donc on fait une *modification* de `L` (qui ne change pas son `id` !), on modifiera *aussi* `GB2` !

```
L[0]=4
print(Grosse_bete[0][0])
print(GB2[0][0])
```

1.3 Comment faire une copie *vraiment* autonome ? A deep copy

Il y a plusieurs méthodes. Il serait amusant de programmer cela en descendant les ramifications de la mémoire. En PYTHON le module `copy` est disponible pour le faire avec la commande `deepcopy`.

```
from copy import deepcopy
GB3=deepcopy(Grosse_bete)
L[0]=17
print(Grosse_bete[0][0])
print(GB3[0][0])
```

2 Les listes comme argument de fonctions

2.1 Rappel du chapitre 4 : une règle d'or

On a vu au chapitre 4, la règle d'or suivante :

Une fonction ne modifie pas ses arguments

Cela signifie que par exemple si on fabrique une fonction `ajoute_un` comme suit :

```
def ajoute_un(a):
    a=a+1  # affectation qui crée un a local ayant pour contenu celui de la valeur a
    #passée en argument plus 1.
    return a # retour de la valeur du a local
```

Si on appelle cette fonction comme suit :

```
x=2
print(ajoute_un(x))
print(x)
```

on a bien eu une *valeur de retour de la fonction* qui vaut 3, mais `x` n'a pas été modifié. On a vu que la solution était de *réaffecter dans* `x` *la valeur de retour de la fonction* autrement dit :

```
x=ajoute_un(x)
```

2.2 Une version plus exacte de la règle d'or :

La vraie règle d'or sur ce problème est la suivante :

Une fonction ne peut modifier que ses arguments qui sont des variables modifiables (mutables) : pour nous pour l'instant, les seules variables modifiables connues sont les *listes*.

2.2.1 Qu'est ce que cela veut dire que le type liste est modifiable (mutable) ?

Cela signifie simplement qu'on peut, comme on l'a expliqué au chapitre précédent, modifier une liste `L` sans la réaffecter, comme on l'a vu ci-dessus, par exemple par `L[0]=` ou `L[0:3]=`

La liste n'a pas changé d'adresse mémoire, elle n'a pas été réaffectée, et en même temps, elle a été vraiment modifiée.

Il faut distinguer une modification `L[i]=`, ou `L[1:4]=` ou `L.append()` d'une affectation `L=`, spécialement à l'intérieur des fonctions.

2.2.2 Deux comportement bien différents

```
def essai1(L):
    if L!=[]:
        L[-1]="fin"

def essai2(L):
    L=L[0:len(L)-1]+["fin"]
```

Dans `essaie2` l'affectation `L=` crée un `L local !`.
Conclusion :
De même on voit apparaître une différence entre :

```
def end1(L):
    L.append("fin")

def end2(L):
    L=L+["fin"]
```

2.2.3 Nous avons déjà fabriqué des fonctions qui modifient la liste en argument !

Dans le T.P. 3, on demandait de créer une fonction `mon_reverse`, qui fasse la même chose que la méthode `reverse` vendue avec la classe liste (cf. le `help(list)`).
Une solution est la suivante :

```
def mon_reverse(L):
    josephine=L[:]# cree une shallow copy de L,
    for i in range(len(L)):
        L[i]=josephine[len(L)-i-1]
```

Point essentiel : dans ce script, il n'y nulle part de `L=`. Une telle commande créerait un `L` local ! En revanche il y a un `L[i]=` qui modifie l'entrée `i` du `L` passé en argument. Voir aussi le corrigé du T.P. 3. par exemple pour le `del`.